POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

SMBUD Project - Spark

Author(s): **Gabriele Ginestroni**
Giacomo Gumiero
Lorenzo Iovine
Nicola Landini
Francesco Leone

Group Number: **10**

Academic Year: 2022-2023

Contents

1 | Introduction

In this chapter will be presented the problem specification and the hypothesis under which the database is implemented.

2 | Data Structure

2.1. Article structure

In the following JSON file we can see how the *Article* is structured in our dataset.
Note: in some fields we decided not to insert all the content only for ease of read reasons.


```
{
"_id": "53e99f86b7602d9702859fdf",
"title": "Locality Sensitive Outlier Detection: A ranking driven
   approach",
"authors": [
    {
    "idAuth": "542a4c9fdabfae61d496694e",
    "org": "Computer Science and Engineering Department, The Ohio State
   University, USA"
    },
    {
    "idAuth": "53f48bc5dabfaea7cd1cce1d",
    "org": "Computer Science and Engineering Department, The Ohio State
   University, USA"
    },
    {
    "idAuth": "53f44b6fdabfaec09f1dd00d",
    "org": "Computer Science and Engineering Department, The Ohio State
   University, USA"
    }
],
"n_citation": 60,
"abstract": "Outlier detection is fundamental to a variety of database
   and analytic tasks. Recently, distance-based outlier detection has
   emerged as a viable and scalable alternative to traditional
   statistical and geometric approaches. In this article we explore the
   role of ranking for the efficient discovery of distance-based
   outliers from large high dimensional data sets. Specifically, we
   develop a light-weight ranking scheme that is powered by locality
```

```
    sensitive hashing, which reorders the database points according to
    their likelihood of being an outlier. We provide theoretical
    arguments to justify the rationale for the approach and subsequently
    conduct an extensive empirical study highlighting the effectiveness
    of our approach over extant solutions. We show that our ranking
    scheme improves the efficiency of the distance-based outlier
    discovery process by up to 5-fold. Furthermore, we find that using
    our approach the top outliers can often be isolated very quickly,
    typically by scanning less than 3% of the data set.",
"doi": "10.1109/ICDE.2011.5767852",
"keywords": [
    "database point",
    "ranking scheme",
    "geometric approach",
    ...
],
"isbn": "978-1-4244-8958-9",
"page_start": "410",
"page_end": "421",
"year": 2011,
"fos": [
    "Locality-sensitive hashing",
    "Anomaly detection",
    "Data mining",
    ...
],
"venue": {
    "raw": "ICDE",
    "type": 0
},
"references": [
    "53e99fddb7602d97028b7e65"
    ]
}
```

2.2. Author structure

In the following `JSON` file we can see how the *Author* is structured in our dataset.
Note: in some fields we decided not to insert all the content only for ease of read reasons.


```
{
"_id":"53f45775dabfaee4dc8162e6",
"name":"Guillermo Jorge-Botana",
"articles":["53e99f86b7602d970285a187"],
"orcid":"0000-0001-5879-6783",
"bio": "Qing-Long Han received the B.Sc. degree in mathematics from the
   ...",
"email":"Guillermo.Jorge-Botana@yahoo.com",
"nationality":"de",
"dob":"1945-06-19T00:00:00Z"
},
```

3 | Dataframe Structure

3.1. Article Structure

```
root
|-- _id: string (nullable = true)
|-- title: string (nullable = true)
|-- authors: array (nullable = true)
|    |-- element: struct (containsNull = true)
|    |    |-- idAuth: string (nullable = true)
|    |    |-- org: string (nullable = true)
|-- n_citation: integer (nullable = true)
|-- abstract: string (nullable = true)
|-- doi: string (nullable = true)
|-- keywords: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- isbn: string (nullable = true)
|-- page_start: string (nullable = true)
|-- page_end: string (nullable = true)
|-- year: integer (nullable = true)
|-- fos: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- references: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- issue: string (nullable = true)
|-- volume: string (nullable = true)
|-- publisher: string (nullable = true)
|-- venue_raw: string (nullable = true)
```

The structure just shown represents an *Article*; its attributes are:

- **_id** is the identifier of a publication.
- **title** represents the title of the publication.
- **authors** is an array that contains: `idAuth` of the authors of the article and the `org` field which represent the affiliation.

- **n_citation** is the number of times that the publication has been mentioned.
- **abstract** is a string containing a brief summary of the contents of the paper.
- **doi** Digital Object Identifier is a persistent and standardized identifier.
- **keywords** is an array containing keywords of the publication.
- **isbn** is an identification code of the venue of the publication.
- **page_start** defines the starting page of the publication.
- **page_end** defines the last page of the publication.
- **year** represents the year of publication.
- **fos** is an array containing the fields of study of the publication.
- **references** set of ObjectIds of the referenced articles.
- **issue** refers to how many times a periodical has been published during that year.
- **volume** is the volume of the venue in which the article has been published.
- **publisher** is the name of the publisher of the article.
- **venue_raw** is the name or the abbreviation of the venue (regardless the year, issue or volume) in which the publication was presented.

3.2. Author Structure

```
root
|-- _id: string (nullable = true)
|-- name: string (nullable = true)
|-- nationality: string (nullable = true)
|-- articles: array (nullable = true)
|    |-- element: string (containsNull = true)
|-- bio: string (nullable = true)
|-- email: string (nullable = true)
|-- orcid: string (nullable = true)
|-- dob: timestamp (nullable = true)
```

The structure just shown represents an *Author*; its attributes are:

- **_id** is the identifier of an author.
- **name** is the name of the author.

- **nationality** is the nationality of the author.
- **articles** is a set of articles identifier of the publications of the author.
- **bio** is a string that describes the author.
- **email** is the email address of the author.
- **orcid** Open Researcher and Contributor ID is a unique identifier for authors of scientific articles.
- **dob** is the birth date of the author.

3.3. Venue Structure

```
root
|-- raw: string (nullable = true)
|-- type: integer (nullable = true)
|-- artIds: array (nullable = false)
|    |-- element: string (containsNull = false)
|-- city: string (nullable = true)
```

The structure just shown represents a *Venue*. This dataframe was obtained using data imported from the article `JSON` shown before. *Venue* attributes are the following:

- **raw** is the name or the abbreviation of the venue (regardless the year, issue or volume) in which the publication was presented.
- **type** indicates the type of the publication.
- **artIds** is a set of articles identifier associated to the venue.
- **city** represents the location of the venue.

4 | Commands and Queries

4.1. Commands

We have identified the following `INSERT` and `UPDATE` commands to show the system basic functionalities.

4.1.1. Insert a new author

Assuming it is not present in the dataset, we created a new row with the values for the new author and we added it to the dataframe.

```
new_author = Row(
    _id="638db170ae9ea0d19fad7a79",
    name="Emanuele Delle Valle ",
    nationality="it",
    articles=[],
    bio="Emanuele Della Valle holds a PhD in Computer Science from the \
        Vrije Universiteit Amsterdam and a Master degree in Computer
                                    Science\
        and Engineering from Politecnico di Milano. He is associate
                                    professor\
        at the Department of Electronics, Information and Bioengineering
                                     of\
        the Politecnico di Milano.",
    email="emanuele.dellavalle@gmail.com ",
    orcid="0000-0002-5176 -5885",
    dob= datetime.strptime("March 7, 1975", "%B %d, %Y")
)

df_authors = df_authors.union(spark.createDataFrame([new_author], schema
                                    = schemaAuthors))
```

4.1.2. Insert a new author

Assuming it is not present in the dataset, we created a new row with the values for a new article written by the author created in section 4.1.1. In order to set the authors, we instantiated an array `new_authors`.

```
new_authors =  [Row("638db170ae9ea0d19fad7a79", "Politecnico di Milano")
                                    , Row("638db170ae9ea0d19fad7a7a", "
                                    Politecnico di Milano")]

new_article = Row(
    _id="638db237d794b76f45c77916",
    title="An extensive study of C-SMOTE, a Continuous Synthetic
                                    Minority Oversampling Technique for
                                    Evolving Data Streams",
    authors=new_authors,
    n_citation=3,
    abstract = "Streaming Machine Learning (SML) studies algorithms that
                                     update their models,\
        given an unbounded and often non-stationary flow of data
                                    performing a single pass. Online \
        class imbalance learning is a branch of SML that combines the
                                    challenges of both class imbalance\
        and concept drift. In this paper, we investigate the binary
                                    classification problem by
                                    rebalancing\
        an imbalanced stream of data in the presence of concept drift,
                                    accessing one sample at a time.",
        doi="10.1016/j.eswa.2022.116630",
    keywords=["Evolving Data Stream","Streaming","Concept drift","
                                    Balancing"],
    isbn="123-4-567-89012-3",
    page_start="39",
    page_end="46",
    year=2022,
    fos=["Computer Science","Stream Reasoning","Big Data"],
    references=["53e99fe4b7602d97028bf743","53e99fddb7602d97028bc085"],
    issue="1",
    volume="196",
    publisher="Elsevier",
    venue_raw="ESA"
)

```

```
df_articles = df_articles.union(spark.createDataFrame([new_article]))
```

4.1.3. Insert a new venue

Assuming it is not present in the dataset, we created a new row with the values for a new venue *ESA* hosted in *Montreal*.
Note: in field `artIds` we set the article created in section 4.1.2.

```
new_venue = Row(
    raw="ESA",
    type=1,
    artIds=["638db237d794b76f45c77916"],
    city="Montreal"
)

df_venues = df_venues.union(spark.createDataFrame([new_venue]))
```

4.1.4. Insert a new article in his author dataframe

Through this command we inserted the article created in section 4.1.2 to its authors. In order to do that, we selected the authors through the ids and we add the article id to their field `articles`.
Note: one of the author is the one created in section 4.1.1.

```
df_authors = df_authors.withColumn(
    "articles",
    f.when(f.col("_id") == "638db170ae9ea0d19fad7a79",
        f.array_union(df_authors.articles, f.array(f.lit("
                                    638db237d794b76f45c77916"))))\
    .when(f.col("_id") == "638db170ae9ea0d19fad7a7a",
        f.array_union(df_authors.articles, f.array(f.lit("
                                    638db237d794b76f45c77916"))))
    .otherwise(f.col("articles"))
)
```

4.1.5. Update the number of citations of referenced publications

Through the following snippet of code is possible to increment the `n_citations` field of the *Publications* referenced by the article created in section 4.1.2.

Note: field `n_citations` is updated for both the referenced articles.

```
df_articles = df_articles.withColumn(
    "n_citation",
    f.when(f.col("_id") == "53e99fe4b7602d97028bf743",
        df_articles.n_citation+1) \
    .when(f.col("_id") == "53e99fddb7602d97028bc085",
        df_articles.n_citation+1)
    .otherwise(f.col("n_citation"))
)
```

4.2. Queries

We have identified the following queries in order to show the system's basic functionalities. In the following sections title we wrote the basic requirements for every query, that, for ease of read, are represented as `SQL` clauses.

4.2.1. Query 1 - WHERE, JOIN

This query returns the type of the venue of an article with a the following title: *"Locality Sensitive Outlier Detection: A ranking driven approach"*.

```
df_articles.join(df_venues, df_articles.venue_raw == df_venues.raw, "
                                    inner")\
           .filter(f.col("title") == "Locality Sensitive Outlier
                                    Detection: A ranking driven approach
                                    ").select("title", "raw", "type").
                                    show()
```

4.2.2. Query 2 - WHERE, LIMIT, LIKE

This query returns the articles whose title string contains *"Machine Learning"*.

```
df_articles.filter(f.col("title").like("%Machine Learning%")).limit(3).
                                    show()
```

4.2.3. Query 3 - WHERE, IN, NESTED QUERY

This query finds authors that has the same nationality of at least one of the authors of *"Locality Sensitive Outlier Detection: A ranking driven approach"* article.
Description: we started creating the list of nationalities of the article's authors adding them to a list: `nationality_list`. Then we look for authors with the a nationality contained in the list.

```
nationalities_list = df_articles.filter(f.col("title") == "Locality
                                    Sensitive Outlier Detection: A
                                    ranking driven approach")\
                                .select(f.explode(df_articles.authors.
                                    idAuth).alias("idAuth"))\
                                .join(df_authors, on=f.col("idAuth") ==
                                    df_authors._id)\
                                .select("nationality")\
                                .agg(f.collect_set("nationality")).
                                    collect()[0][0]

df_authors.filter(f.col("nationality")\
          .isin(nationalities_list)).limit(10).show()
```

4.2.4. Query 4 - GROUP BY, JOIN, AS

This query finds the 3 most frequent keywords of articles written by italian authors.
Description: we started from finding all the italian authors, then we grouped them by articles. Then we count the keywords putting them in descending order.

```
df_italian = df_authors.filter(f.col("nationality") == "it")\
                       .select(f.explode("articles")).withColumnRenamed(
                                    "col","articles")
df_italian = df_italian.groupby("articles").count()

df_keywords = df_italian.join(df_articles, df_italian.articles ==
                                    df_articles._id, "inner")\
                        .select("articles", f.explode("keywords")).
                                    withColumnRenamed("col","keywords")\
                        .groupby("keywords")\
                        .agg(f.count("keywords").alias("n_occurences"))\
                        .sort("n_occurences", ascending=False)\
                        .limit(3).show()
```

4.2.5. Query 5 - WHERE, GROUP BY

This query finds the cities with more than 65 venues.

```
df_venues.groupby("city")\
         .count()\
         .filter(f.col("count") > 65)\
         .sort("count", ascending=False).show()
```

4.2.6. Query 6 - GROUP BY, HAVING, AS

This query finds the field of studies that appears more than 15 times.

```
df_articles.select("_id", "title", f.explode("fos")).withColumnRenamed("
                                    col", "fos")\
           .groupby("fos")\
           .agg(f.count("fos").alias("n_occurence"))\
           .filter(f.col("n_occurence") > 15).show()
```

4.2.7. Query 7 - WHERE, GROUP BY, HAVING, AS

This query finds all the volumes with at least 5 articles in the dataset, published after 2000.

Description: we started from filtering the year of publication, then we grouped by `venue_raw` and `volume` in order to separate different edition of the same venue. After that we filtered the number of articles for every edition.

```
df_articles.filter(f.col("year") > 2000)\
           .groupby("venue_raw", "volume")\
           .agg(f.count("volume").alias("num_articles"))\
           .filter(f.col("num_articles") > 4)\
           .show()
```

4.2.8. Query 10 - WHERE, GROUP BY, HAVING, 2 JOINS

This query finds all the authors that published on more than 2 Journals.

```
df_exploded_authors = df_authors.alias("auth")\
```

```
2                                   .select("auth._id","auth.name", f.
                                        explode("auth.articles").alias("
                                        article"))\
3                                   .join(df_articles.alias("art"), on=f.col
                                        ("article") == df_articles._id)\
4                                   .select("auth._id","auth.name","art._id"
                                        ,"art.venue_raw")\
5                                   .join(df_venues.alias("ven"), on=f.col("
                                        venue_raw") == df_venues.raw)\
6                                   .filter(f.col("type") == 1)\
7                                   .groupBy("auth._id")\
8                                   .agg(f.first("name").alias("name"),f.
                                        countDistinct("raw").alias("
                                        venue_count"),f.concat_ws(" - ",f.
                                        collect_set("raw")).alias("
                                        venues_list"))\
9                                   .filter(f.col("venue_count") > 2)\
10                                  .orderBy("venue_count", ascending=False)
                                        .show(3,truncate=False)
```